NASA-TM-112357

/ID-60-TM
018251

Unfavorable Strides in Cache Memory Systems

David H. Bailey

RNR Technical Report RNR-92-015
May 21, 1992

NASA
National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

ARC 275a (Feb 81)

Unfavorable Strides in Cache Memory Systems

David H. Bailey

RNR Technical Report RNR-92-015
May 21, 1992

Abstract

An important issue in obtaining high performance on a scientific application running on a cache-based computer system is the behavior of the cache when data is accessed at a constant stride. Others who have discussed this issue have noted an odd phenomenon in such situations: a few particular innocent-looking strides result in sharply reduced cache efficiency. In this paper, this problem is analyzed, and a simple formula is presented that accurately gives the cache efficiency for various cache parameters and data strides.

The author is with the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

Introduction

Scientists accustomed to running large computationally intensive applications on Cray supercomputers have never had to concern themselves with cache issues. However, with the recent sharp rise in the floating point performance of RISC workstations, many scientists are now using these systems for serious computations, and cache issues can no longer be avoided. Another avenue from which supercomputer scientists have been introduced to cache memories is the recent incorporation of RISC processors into highly parallel supercomputers.

Many important scientific applications do not feature exclusively stride one data access but instead feature large nonunit strides. For instance, many codes perform similar operations on each dimension of a two or three dimensional array. Performing computations in the first dimension can be done with unit stride, but the strides of the computations in the other dimensions are typically large values, and significantly degraded performance may result when the codes are ported to cache-based systems without change.

One solution to this problem is to rewrite the code to employ array transpositions between the computational steps in each dimension. In this way all computation can be done at unit stride. But such revision may require substantial effort, and it may still not result in significant performance improvement unless the time spent in stride one computation is substantial enough to offset the cost of the array transpositions.

As a result, many scientists ignore the problem, accepting with a certain fatalism that their codes will not perform well in the nonunit stride stages of the computation. However, for some programs the reduction in performance is so severe that programmers are willing to expend the necessary effort to understand and alleviate this problem.

1. Definitions and Notation

To better understand the phenomenon of performance reduction with strides, consider the following model. Let the cache be configured as $R = 2^r$ rows of cache lines, and assume that each cache line contains $W = 2^w$ words.

It will be assumed that this cache memory system operates as follows. When a word at an address A is fetched, it is placed in row Q, where Q is determined by zeroing the bits in the address to the left of the rightmost $r + w$ bits and then shifting the resulting integer to the right by w bits (i.e. dividing by W). Note that this operation produces an integer Q in the range $0 \leq Q < R$. When a single word is requested, all W words of the W-long cache line that it resides in are actually fetched.

If the stride S of a vector fetch is unity, then W consecutive words reside on the same cache line, and only that one cache line needs to be physically fetched. This is obviously a very favorable situation. The situation is similarly quite favorable if the memory stride is some integer less than W, since in that case many cache lines contain multiple words required by the CPU. Many scientific applications, however, involve strides larger than W, so that each cache line retrieved from memory contains at most one word required by the CPU. This second case will be the focus of this paper.

Many cache-based systems employ "associativity sets", which are multiple cache lines on each row. They operate as follows. Suppose a request is made for data at an address A that lies (by the address masking operation described above) in the same row as a cache line previously requested. Then one of the C cache lines (usually the "oldest") on that row is replaced by the requested cache line (this will be termed a cache "flush"). In this way, even if only one word per cache line is required by the CPU, potentially RC words may be cached.

Unfortunately, at some strides even RC words cannot be cached because some rows are overutilized, while other rows are underutilized. Let us consider a vector fetch of L words with stride S and ask what fraction of the L resulting cache lines remain in the cache when the fetch is complete. This question is of interest for two reasons: (1) a computation may need to access this same set of L words again, and (2) if this vector fetch was a single row of a matrix stored in column major order, the next W rows of the matrix reside in these same cache lines. Either way, performance will be significantly improved if these cache lines can remain in the cache.

Accordingly, the efficiency E of a vector fetch of length L will be defined as T/L, where T is the number of cache lines that still remain in the cache when the vector fetch operation is complete, and where L is the vector length. For simplicity, in the following it will be assumed that $L = RC$.

An obvious example of an inefficient stride is a large power of two. Then all cache lines will be fetched into the same row of the cache, and the other $R-1$ rows will be completely unutilized. The resulting efficiency is only $1/R$. Clearly if an application program has arrays whose dimensions are large powers of two, these arrays should be "padded" by declaring their leading dimensions to be slightly larger than a power of two. Most users of Cray systems are familiar with this tuning technique, since it eliminates bank conflicts that may reduce performance by factors as high as 10 or 20.

2. Cache Efficiency with Non-Power-of-Two Strides

It comes as a surprise to many scientists accustomed to Crays that large power of two strides are not the only particularly unfavorable strides for cache memory systems [3]. To facilitate more concrete discussion in the following, we will consider the particular case $R = 32$, $C = 4$ and $W = 16$. These values match the cache parameters of the IBM RS 6000/320 system. We will also assume in the following that the vector length L of the fetch is 128.

When $S = 72$, it turns out that in 128 consecutive fetches, the respective cache lines neatly fill the 32×4 array, resulting in perfect utilization of the cache (except that only one word in each cache line may actually be required by the CPU). The resulting efficiency E is unity, even though 72 is divisible by eight, a highly unfavorable situation on many vector computers. Now consider $S = 73$, a completely favorable stride for most vector computers. In this case the cache efficiency is only 0.414. The efficiencies for strides 16 to 256 are shown in Figure 1. This is obviously a very complicated function.

This curious phenomenon has been noted by others [1, 2, 3, 5]. One way to understand it is to list the row numbers of consecutively fetched cache lines in a 128-long vector fetch,

4	9	13	18	22	27	31
4	9	13	18	22	27	31
4	9	13	18	22	27	31
4	8	13	18	22	27	31
4F	8	13F	18F	22F	27F	31F
4F	8	13F	17	22F	27F	31F
4F	8	13F	17	22F	27F	31F
4F	8F	13F	17	22F	26	31F
4F	8F	13F	17	22F	26	31F
4F	8F	13F	17F	22F	26	31F
3	8F	13F	17F	22F	26	31F
3	8F	13F	17F	22F	26F	31F
3	8F	12	17F	22F	26F	31F
3	8F	12	17F	22F	26F	31F
3F	8F	12	17F	21	26F	31F
3F	8F	12	17F	21	26F	31F
3F	8F	12F	17F	21	26F	30
3F	8F	12F	17F	21	26F	30
3F	8F					

Table 1: Row Numbers for Successive Fetches When $S = 73$

with stride 73, in a seven-wide table (see Table 1). This table also includes the notation F to indicate instances when a cache flush would occur. It is clear from examining this table that the root cause of this poor performance is the very nearly periodic behavior of these row numbers.

Recall that address bits higher than position $r + w$ are ignored when placing the cache line in a row. Thus we may in general write the row number Q of the k-th word fetched as

$$Q(k) \;=\; \mathrm{int}\left[\frac{1}{W} \bmod (kS, RW)\right]$$

where int denotes the greatest integer function, and where mod denotes the modulo operation with results in the range between zero and the second argument minus one. The function $Q(k)$ is precisely periodic with period RW. But when the stride S is exactly (or very nearly) a simple fraction of RW, then this function is also precisely (or very nearly precisely) periodic with period $\mathrm{nint}(RW/S)$, where nint is the nearest integer function.

In this example, $RW = 512$ and $S = 73$. Indeed, the fraction $Q = 73/512$ is very close to the simple fraction $1/7$. In fact, $7 \times 73 = 511$, so that every seventh value of $\mathrm{mod}(kS, RW)$ differs by only one, and when divided by 16 the resulting row numbers are identical for 16 consecutive columns. But a string of 16 consecutive identical row numbers results in 12 flushes, since only four of these can be accommodated in a single row of the cache matrix. Thus once the initial repeating rows of the cache table are filled, the remaining fetches will produce a flush approximately 75 percent of the time.

From these facts one can easily compute the approximate cache efficiency E for this example. In Table 1, the first $4 \times 7 = 28$ members of the sequence completely fill cache rows 4, 9, 13, 18, 22, 27 and 31, except that row nine has one line empty. Thus we have the approximation

$$E = \frac{128 - 0.75 \times (128 - 28)}{128} = 0.414$$

which in this case exactly matches the actual efficiency determined by counting flushes in Table 1.

As we have seen, the ratio $G = 0.75$ used in the above calculation results from the fact that $7 \times 73 = 511$ differs from 512 by only one. When this difference D is zero (i.e. when S is a large power of two, such as 64), then the corresponding value of G may easily be seen to be unity. When this difference is two, $G = 1/2$; when the difference is three, $G = 1/4$; and when the difference is four or more, $G = 0$. In general, it can been shown that G is given by the formula

$$G = \frac{1}{C} \max(C - D, 0)$$

Suppose that $S/(RW)$ is very close to a simple fraction a/b, $b \leq R$, so that $D = |bS - aRW|$ is small. Compute G from the above formula. Now we may write a formula that is an approximation to the cache efficiency E for general strides and cache parameters:

$$E = \frac{L - G(L - bC)}{L}$$

A graph of the efficiencies for various strides in the standard case used above, computed with the above formula, is shown in Figure 2. By comparing Figures 1 and 2, it is clear that this formula is very accurate, particularly at the "spikes", which are the cases of greatest interest. In fact, the flush count $F = G(L - bC)$, which is the key subexpression of this formula, is (with one exception) always within one of the actual value whenever G is nonzero.

3. A Random Stride Approximation

When the difference D is greater than C, the formula above gives perfect efficiency, since G in that case is zero. However, the actual efficiency is somewhat less than unity for many such cases, resulting in a low-level background "noise" (compare Figures 1 and 2). This phenomenon can be explained by nothing that when the stride S is a substantial fraction of RW, the operation $\text{mod}(kS, RW)$ is a good pseudorandom number generator, and a certain number of "collisions" can be expected to occur in the resulting row numbers. In fact, this operation is a member of the widely studied class of linear congruential pseudorandom number generators ([4], p. 9).

If one assumes that the assignment of memory fetches to the R rows is actually random, then one can compute the expected cache efficiency by applying techniques of probability

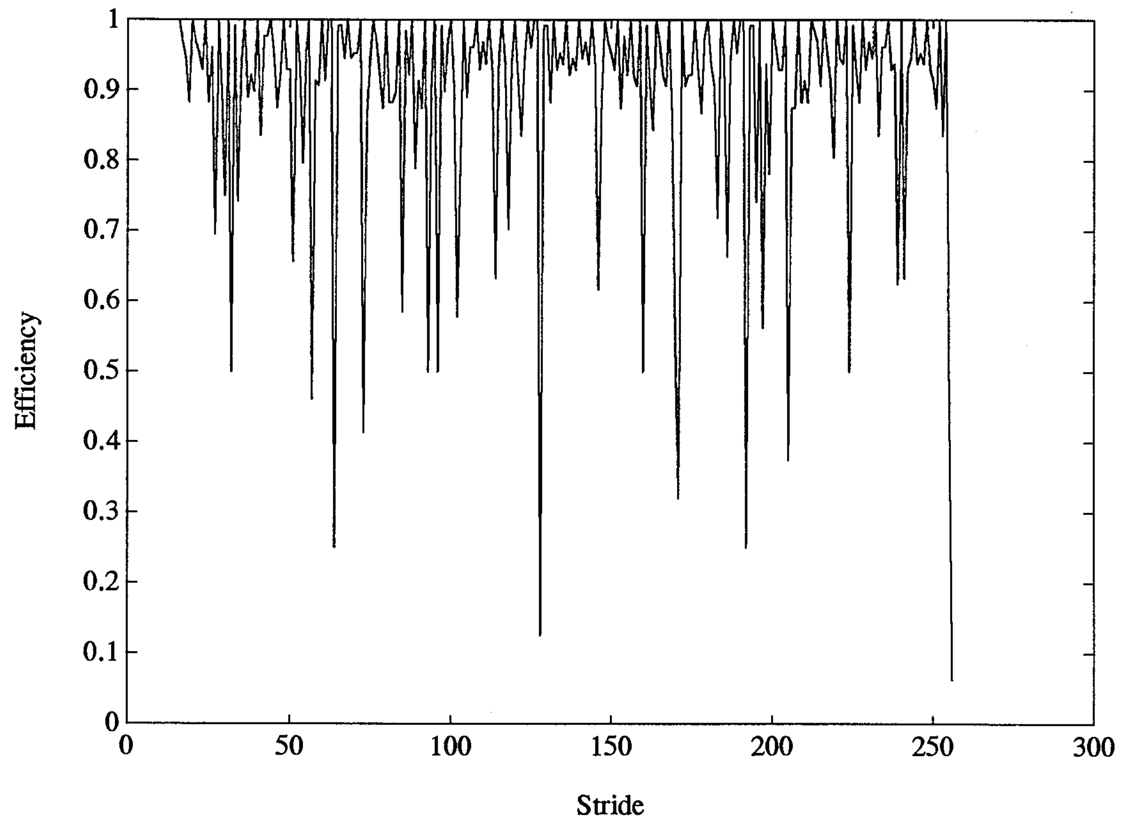

Figure 1: Cache Efficiencies for Various Strides

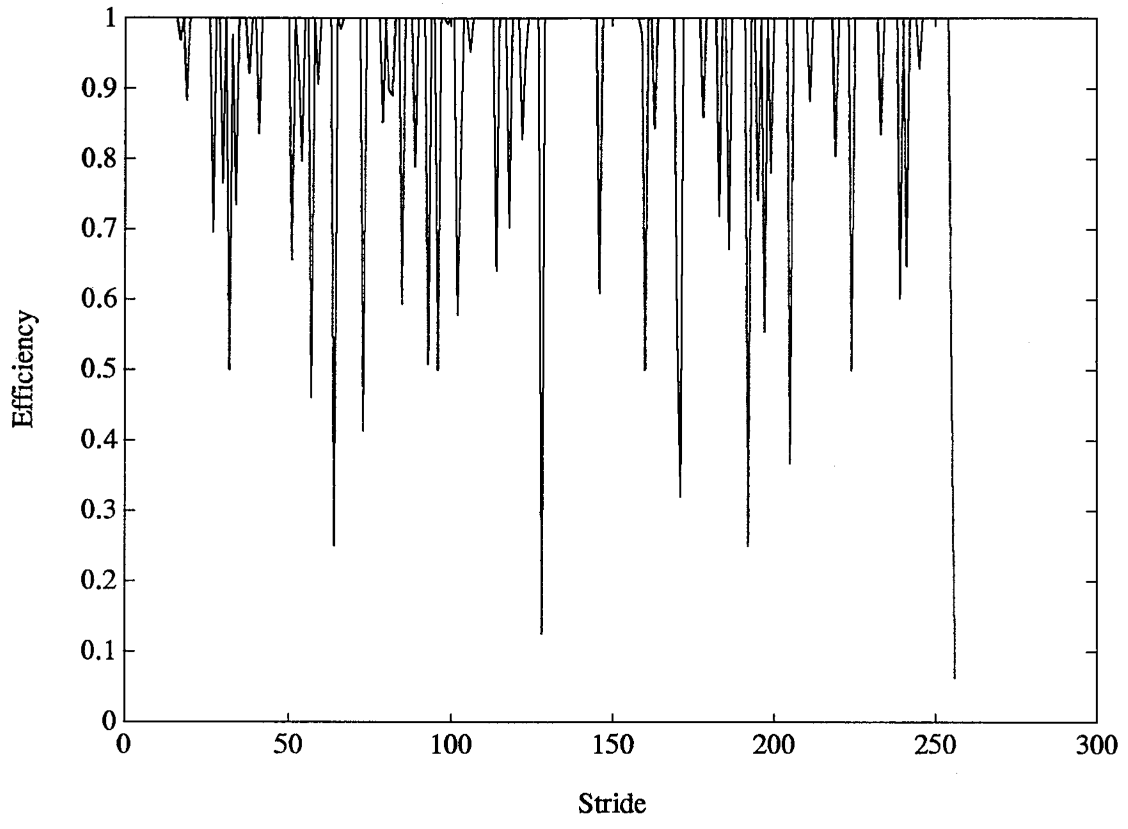

Figure 2: Cache Efficiencies Using the Formula

and statistics. The probability $P(k)$ that an individual row contains exactly k entries after an L-long fetch is given by the formula for a binomial distribution:

$$P(k) = \binom{L}{k} p^k (1-p)^{L-k}$$

where $p = (R-1)/R$. The expected number of flushes F is then

$$F = R \sum_{k=C+1}^{\infty} (k-C)P(k)$$

and the resulting expected efficiency $E = (L-F)/L$. For the example parameters above, this formula yields $E = 0.808$. The actual average efficiency, determined from the data in Figure 1, is 0.895. This indicates that the operation $\mathrm{mod}(kS, RW)$ actually behaves somewhat better than a true random number generator.

4. Finding Simple Fractions

One detail was omitted from the above discussion: how does one determine the minimum difference D for a given stride, or in other words, how does one determine the best simple fraction approximation a/b to $S/(RW)$?

A straightforward means to find this fraction is by exhaustion, since the periodic effect ceases to exist when b exceeds R (since in that case $bC > L$). In other words, one can compute $D = |bS - aRW|$ for all integers a and b less than R. If the smallest such D is less than C, then the periodic effect exists and the above formulas apply. When R is even moderate in size, however, this procedure is time-consuming.

A more direct and elegant means to find these rational approximations a/b is to employ the Euclidean algorithm ([4], p. 319), as follows. Start with the 2-long vector (S, RW) and the 2×2 identity matrix. At a given step let x be the smaller entry of the 2-long vector, let y be the larger entry, and let X and Y be the columns of the 2×2 matrix corresponding to x and y. Compute $q = \mathrm{int}(y/x)$. Then replace y by $y - qx$ and X by $X + qY$. This process continues until one entry of the vector is zero. At that point one column of the final matrix will contain the original vector (with any common factor divided out) and the other column will contain a close rational approximation. In this application, the Euclidean algorithm may be halted whenever an entry of the matrix exceeds R.

The operation of this algorithm in this application is more easily understood by an example. Let us consider the particular parameters as above, with the stride $S = 197$. In other words, we wish to find a good simple fraction approximation a/b to $197/512$. The algorithm proceeds as shown below. The value of q used in each step (computed from the previous step's vector) is shown at the right.

$$\begin{pmatrix} 197 \\ 512 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 197 \\ 118 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \qquad q = 2$$

$$\begin{pmatrix} 79 \\ 118 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} \qquad q = 1$$

$$\begin{pmatrix} 79 \\ 39 \end{pmatrix} \quad \begin{pmatrix} 2 & 1 \\ 5 & 3 \end{pmatrix} \qquad q = 1$$

$$\begin{pmatrix} 1 \\ 39 \end{pmatrix} \quad \begin{pmatrix} 2 & 5 \\ 5 & 13 \end{pmatrix} \qquad q = 2$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 197 & 5 \\ 512 & 13 \end{pmatrix} \qquad q = 39$$

In this case the desired pair of integers (a, b) is in the next-to-last column generated in the matrix, i.e. (5, 13). Note that $5/13 = 0.38462\cdots$ is indeed an excellent approximation to $197/512 = 0.38477\cdots$.

Here the final column generated, (197, 512), is identical to the original vector. If S is divisible by a power of two, then the final column generated will be the original vector with the common power of two divided out. In that case, and if both entries of the final column are less than or equal to R, then this final column should be selected for (a, b) instead of the previously generated column. If for a given stride S, no pair (a, b), $b \leq R$ is found that satisfies $|bS - aRW| < C$, then the periodic effect does not exist, and the stride may be considered a favorable stride.

5. Improving Cache Performance of Data Access with Strides

What can a programmer do if his or her program features a particularly unfavorable stride? The most straightforward solution is to "pad" (slightly increase) the leading dimensions of arrays having such dimensions. This solution has the advantage that in most cases only dimension statements need to be changed, and the executable part of the program does not need to be altered. A few extra rows of two-dimensional arrays are "wasted" in this manner, but the resulting performance improvement is almost certainly worth the additional memory required.

There does not appear to be a simple formula giving the optimal amount of padding for a given unfavorable stride (i.e. array dimension) S, but in practice it suffices to merely evaluate the efficiency function described above for $S + 1$, $S + 2$, etc. until an efficient stride is found. In examples the author has studied, it appears that a pad of only one or two is effective in most cases.

However, this type of tuning should not be necessary, nor should it be necessary for programmers to analyze whether their strides are unfavorable. By applying techniques such as those described in this paper, compilers should be able to detect unfavorable strides and automatically adjust the appropriate array dimensions. Such adjustments will need to be optional, since they technically depart from the Fortran-77 standard, but they will likely be welcomed by the majority of users who prefer the compiler to shield them from such unsavory features of the underlying architecture.